Corrigé TP6

1. Les bords de u sont les mots ε, a, aba. On a donc $B(u) = aba$.

2. Aucune difficulté si ce n'est qu'il faut réserver sur le tas suffisamment de place pour la sous-chaîne, y compris un emplacement pour le caractère de fin de chaîne $\backslash 0$.

```c
char* sous_chaine(char* m, int i, int j)
{
    int n = j-i+1;
    char* sc = (char*)malloc(sizeof(char)*(n+1));
    for (int k = 0; k < n; k++)
    {
        sc[k] = m[i+k];
    }
    sc[n] = '\0';
    return sc;
}
```

La complexité de cette fonction est $O(j-i)$ puisqu'on alloue et initialise ce nombre de cases.

3. Le bord maximal qu'on souhaite calculer ici est celui de $u = m_0 ... m_{i-1}$. On maintient à jour une variable longueur_bord qui stocke la longueur du plus long bord de u connu à ce jour. On considère toutes les longueurs $l \in [\![1, i-1]\!]$ par ordre croissant (on exclut $l = i$ pour s'assurer de ne considérer que des préfixes et suffixes différents de u), on extrait le préfixe de longueur l de u et le suffixe de longueur l de u, on les compare et s'ils sont égaux, on met à jour longueur_bord.

   Le cas où le seul préfixe-suffixe de u est ε est correctement traité vu l'initialisation de longueur_bord.

```c
int bord_maximal(char* m, int i)
{
    int longueur_bord = 0;
    int longueur_mot = i;
    for (int l = 0; l < longueur_mot-1; l++)
    {
        char* prefixe = sous_chaine(m,0,l);
        char* suffixe = sous_chaine(m,longueur_mot-1-l,longueur_mot-1);
        if (strcmp(prefixe,suffixe) == 0)
        {
            longueur_bord = l + 1;
        }
        free(prefixe);
        free(suffixe);
    }
    return longueur_bord;
}
```

La complexité temporelle de cette fonction est en $O(i^2)$ puisqu'on fait de l'ordre de $2i$ extractions de chaînes qui se font toutes en temps $O(i)$ d'après la question précédente.

4. On utilise simplement bord_maximal pour chacun des $|m|+1$ préfixes de m :

```
int* bords_maximaux(char* m)
{
    int n = strlen(m);
    int* bords = (int*)malloc(sizeof(int)*(n+1));
    for (int l = 0; l <= n; l++)
    {
        bords[l] = bord_maximal(m,l);
    }
    return bords;
}
```

L'analyse de la question précédente nous informe que la complexité de cette fonction est en

$$O\left(\sum_{i=0}^{|m|} l^2\right) = O(|m|^3)$$

5. a) Soit $u \in \Sigma^+$ et notons $\mathcal{B}(u)$ l'ensemble de ses bords.

Montrons par récurrence que pour tout $k \in \mathbb{N}^\star$, $B^k(u)$ est un bord de u. C'est immédiatement le cas pour $k=1$. De plus, si $v = B^k(u)$ est un bord de u alors c'en est un préfixe et un suffixe donc par transitivité, $B(v)$ est un préfixe de v donc de u et est un suffixe de v donc de u. Donc $B^{k+1}(u)$ est un bord de u.

Comme un bord de u est distinct de u, on a également par récurrence que la suite $(|B^k(u)|)_{k\in\mathbb{N}^\star}$ est strictement décroissante. Comme $|B(u)| \leq |u|-1$, on est donc assuré que pour $k \geq |u|$, on a $B^k(u) = \varepsilon$. Les deux points précédents montrent qu'il existe $k \leq |u|$ tel que $\{B^l(u) \,|\, l \in [\![1,k]\!]\} \subset \mathcal{B}(u)$.

Montrons à présent l'inclusion réciproque. Si w est un bord de u, on peut encadrer sa taille de la façon suivante : $|B^i(u)| \leq |w| \leq |B^{i-1}(u)|$ pour $i \in [\![2,k]\!]$. Si $|w| = |B^{i-1}(u)|$, comme ces deux mots sont préfixes de u alors ils sont égaux ce qui conclut. Sinon, w est un bord de $B^{i-1}(u)$ donc $|w| \leq |B(B^{i-1}(u))| = |B^i(u)|$ par définition d'un bord maximal. Dans ce cas, w et $B^i(u)$ sont deux préfixes de u de même taille donc sont égaux et on conclut à nouveau.

b) Un bord de ua est soit égal à ε, soit s'écrit wa avec w un bord de u. D'après la question précédente, les bords de ua se trouvent donc parmi les éléments de l'ensemble $B = \{B(u)a, B^2(u)a, ..., B^k(u)a, \varepsilon\}$.

Tous les éléments de B sont des suffixes de ua mais tous n'en sont pas nécessairement des préfixes. Ainsi, les bords de ua sont les éléments de B qui sont préfixes de ua et le plus grand d'entre eux est par définition le bord maximal de ua, comme annoncé.

6. On traduit le pseudo-code fourni dont la correction repose sur la question 5 :

```
int* bords_maximaux_dyn(char* m)
{
    int n = strlen(m);
    int* bords = (int*)malloc(sizeof(int)*(n+1));
    bords[0] = 0;
    bords[1] = 0;
    for (int i = 2; i <= n; i++)
    {
        int longueur_bord_i = bords[i-1];
        while ((longueur_bord_i > 0) && m[longueur_bord_i] != m[i-1])
        {
```

```
            longueur_bord_i = bords[longueur_bord_i];
        }
        //Si on est sorti grâce à la deuxième condition, c'est qu'on a réussi à étendre un bord précédent avec la dernière lettre du préfixe de taille i
        if (m[longueur_bord_i] == m[i-1]) bords[i] = longueur_bord_i +1;
        //Sinon c'est que le bord maximal pour le préfixe de taille i est epsilon
        else bords[i] = 0;
    }
    return bords;
}
```

7. Pour mieux voir ce qui se passe dans cet algorithme, on en modifie légèrement le pseudo-code comme suit. Les lignes 8 à 11 peuvent être réécrites sans modifier la complexité en :

   Si $m_l = m_{i-1}$
       $l \leftarrow l + 1$
   $L[i] \leftarrow l$

   En effet, si on a l'égalité $m_l = m_{i-1}$ au sortir de la boucle tant que de la ligne 5, il faut écrire $l+1$ dans la case $L[i]$ et sinon, la condition de sortie de la boucle tant que assure que $l = 0$ et c'est justement la valeur qu'il faudra mettre dans $L[i]$ dans ces conditions.

   En mettant ainsi l à jour, cette variable contient en fin de boucle pour exactement la valeur qui lui sera assignée en entrée de la boucle pour suivante selon le pseudo-code initial. On peut donc réécrire le pseudo-code de l'énoncé (sans en modifier la complexité asymptotique puisqu'elle sera au moins en $O(|m|)$) :

   1. bords_maximaux(m) =
   2.   Initialiser un tableau L de taille $|m| + 1$ rempli de 0
   3. $l \leftarrow 0$
   4.   Pour i allant de 2 à $|m|$
   5.   Tant que $l > 0$ et $m_l \neq m_{i-1}$
   6. $l \leftarrow L[l]$
   7.   Si $m_l = m_{i-1}$
   8. $l \leftarrow l + 1$
   9. $L[i] \leftarrow l$
   10.   Renvoyer L

   On remarque ensuite deux choses :

   - Passer dans la boucle tant que de la ligne 5 fait décroître strictement l.
   - La seule façon d'incrémenter l est en ligne 8 et une telle incrémentation ne peut arriver qu'une seule fois par itération de la boucle pour. Ainsi, l est incrémenté au plus $|m| - 1$ fois de une unité.

   Pour le prouver rigoureusement, on peut montrer par exemple que la propriété "$l < i - 1$ et pour tout $k \in]\!]0, i[\![$, $L[k] < k$" est un invariant pour la boucle pour. Comme l reste toujours positif (à cause de la première condition de sortie de la boucle tant que et de l'initialisation de l), s'il croît au plus $|m| - 1$ fois de 1 au cours de la boucle pour, il ne peut décroître que $|m| - 1$ fois **en tout** au cours de la boucle pour. Donc on passe en tout moins de $|m| - 1$ fois dans la boucle tant que.

   On en déduit immédiatement que la complexité de cet algorithme est en $O(|m|)$.

8. a) Comme # n'apparaît ni dans m ni dans t, les bords maximaux de $m\#t$ valent au plus m. De plus, dès qu'un bord (forcément maximal) de $m\#t$ vaut m, alors m est suffixe de t donc on a trouvé une occurrence de m dans t. Pour détecter les occurrences de m dans t, il suffit donc de calculer les bords

maximaux de $m\#t$: on obtient un tableau et chaque case contenant $|m|$ dans ce tableau donne la position d'une occurrence de m dans t.

b) On commence par écrire une fonction permettant de créer la chaîne $m\#t$. La complexité de cette fonction est évidemment en $O(|m| + |t|)$.

```
char* concatenation(char* m, char* t)
{
    int longueur_m = strlen(m);
    int longueur_t = strlen(t);
    char* c = (char*)malloc(sizeof(char)*(longueur_m + longueur_t + 2));
    for (int i = 0; i < longueur_m; i++)
    {
        c[i] = m[i];
    }
    c[longueur_m] = '#';
    for (int i = 0; i < longueur_t; i++)
    {
        c[longueur_m+1+i] = t[i];
    }
    c[longueur_m + longueur_t +1] = '\0';
    return c;
}
```

La seule difficulté restante est de retrouver la position du début du motif m à partir de i lorsque i indice une case contenant $|m|$ dans le tableau des bords maximaux de $m\#t$.

```
void KMP(char* m, char* t)
{
    char* c = concatenation(m,t);
    int* bords = bords_maximaux_dyn(c);
    int n = strlen(m);
    for (int i = 0; i < strlen(t); i++)
    {
        if (bords[n+i+1] == n)
        {
            printf("Le motif apparaît en position %d \n", i-n);
        }
    }
    free(bords);
    free(c);
}
```

c) D'après la question 7, la complexité temporelle de cette fonction est en $O(|m| + |t|)$, c'est-à-dire linéaire en la taille de l'entrée ! Rappelons que tous les algorithmes de recherche de motif vus l'année dernières sont en $O(|t||m|)$ au pire cas, même si en pratique ils peuvent être efficaces.

Cette fonction ne nécessite que le stockage d'un tableau de taille en $O(|m|)$. L'automate des occurrences quant à lui nécessite un espace de stockage en $O(|m||\Sigma|)$ puisqu'il comporte $|m||\Sigma|$ transitions. Cette implémentation de l'algorithme de Knuth-Morris-Pratt réduit donc la complexité spatiale par rapport à une version qui construirait explicitement l'automate des occurrences tout en conservant une complexité temporelle linéaire en la taille de l'entrée.

d) Alors ? Où ?

9. On montre facilement par double implication que deux mots u et v sont conjugués si et seulement si $|u| = |v|$ et u est un facteur de vv. Pour implémenter sont_conjugués, on introduit donc deux fonctions auxiliaires (dont le code est immédiat) :

   - concat créé une nouvelle chaîne correspondant à la concaténation de ses entrées.
   - KMP_mem a le même fonctionnement que KMP mais plutôt que d'afficher les positions des occurrences du motif m dans t, elle renvoie un booléen indiquant si m apparaît dans t.

```c
bool sont_conjugues(char* u, char* v)
{
    char* v_carre = concat(v,v);
    bool res = (strlen(u) == strlen(v)) && (KMP_mem(u,v_carre));
    free(v_carre);
    return res;
}
```

   La création de v_carre nécessite $2|v|$ copies de caractères, le calcul des longueurs de u et v se fait linéairement en leurs tailles respectives et l'appel à KMP_mem(u,v_carre) nécessite de l'ordre de $|u| + 2|v|$ opérations d'après la question 8c, donc on obtient bien une complexité en $O(|u| + |v|)$.

10. Un mot u contient un facteur carré si et seulement si il existe un conjugué de u non égal à u qui possède un bord non vide. Ceci se montre par double implication et repose sur le fait que, si $u = u_1 w^2 u_2$ avec $w \neq \varepsilon$, alors w est un bord de wu_2u_1w qui est bien un conjugué non trivial de u.

    On écrit donc une fonction permutation qui construit la chaîne de caractères $u_i u_{i+1} ... u_{k-1} u_0 ... u_{i-1}$ si son entrée est constituée de $u = u_0 ... u_{k-1}$ et i (vous l'avez implémentée dans le DS2 !). Pour chacune des $|u| - 1$ permutations non triviales de u, on calcule en $O(|u|)$ son bord maximal à l'aide de bord_maximaux_dyn. On s'intéresse uniquement à la dernière case du tableau ainsi créé : s'il contient une valeur non nulle, on a trouvé un facteur carré. Si aucune des permutations susmentionnées ne produit de bord maximal non vide, u ne contient pas de facteur carré.

```c
bool carre(char* u)
{
    int n = strlen(u);
    for (int i = 1; i < n; i++)
    {
        char* p = permutation(u,i);
        int* bords = bords_maximaux_dyn(p);
        if (bords[n] > 0)
        {
            free(p);
            free(bords);
            return true;
        }
        free(p);
        free(bords);
    }
    return false;
}
```

11. a) Si v est une période de u, il existe $k \geq 1$ et un préfixe v' de v tel que $u = v^k v'$. Comme $u = vw$, par régularité, $w = v^{k-1}v'$. Comme v' est un préfixe de v, il existe v'' tel que $v = v'v''$. On constate alors que $wv''v' = v^{k-1}v'v''v' = v^{k-1}vv' = v^k v' = u$. Donc w est un préfixe de u, un suffixe de u et est différent de u puisque $v \neq \varepsilon$: c'est un bord de u.

Réciproquement, si w est un bord de u, il existe v' tel que $u = vw = wv'$. Comme $v \neq \varepsilon$, il existe $n \geq 1$ tel que $|v^n| \geq |u|$. Alors $v^n w = v^{n-1}wv' = w^{n-2}w(v')^2 = ... = w(v')^n = wv'(v')^{n-1} = u(v')^{n-1}$. Comme $|u| \leq |v^n|$ par choix de n, cette égalité implique que u est préfixe de v^n donc que v est une période de u.

b) La question précédente montre que la plus petite période de u est le mot v tel que $u = vw$ et w est le bord maximal de u. On calcule donc la longueur de ce bord avec bords_maximaux_dyn ce qui permet d'extraire le mot v désiré, le tout en $O(|u|)$ d'après les complexités établies en 2 et 8c.

```
char* periode(char* u)
{
    int n = strlen(u);
    int* bords = bords_maximaux_dyn(u);
    char* p = sous_chaine(u,0,n-bords[n]-1);
    free(bords);
    return p;
}
```

12. On considère un mot u sur un alphabet Σ. Si $\#$ est une lettre n'intervenant pas dans u, observons le mot $m = u\#\overline{u}$ où $\overline{u}$ est le transposé de u. On constate qu'un préfixe v de u est un palindrome si et seulement si v est un bord de m. La preuve se fait par double implication et repose sur le point suivant : si v est préfixe de u, alors il existe w tel que $u = vw$ et en injectant dans m on a $m = vw\#\overline{w}\,\overline{v}$.

Pour répondre à la question, il suffit donc de calculer les bords de m c'est-à-dire l'ensemble $\{B(m), B^2(m), ..., B^{|m|}(m)\}$ d'après la question 5a ce que le calcul des bords maximaux de m permet d'accomplir.